A Language for Counterfactual Generative Models

Zenna Tavares [1] **James Koppel** [1] **Xin Zhang** [2] **Ria Das** [1] **Armando Solar-Lezama** [1]

Abstract

We present OMEGA$_{\mathrm{C}}$, a probabilistic programming language with support for counterfactual inference. Counterfactual inference means to observe some fact in the present, and infer what would have happened had some past intervention been taken, e.g. "given that medication was not effective at dose x, what is the probability that it would have been effective at dose $2x$?" We accomplish this by introducing a new operator to probabilistic programming akin to Pearl's **do**, define its formal semantics, provide an implementation, and demonstrate its utility through examples in a variety of simulation models.

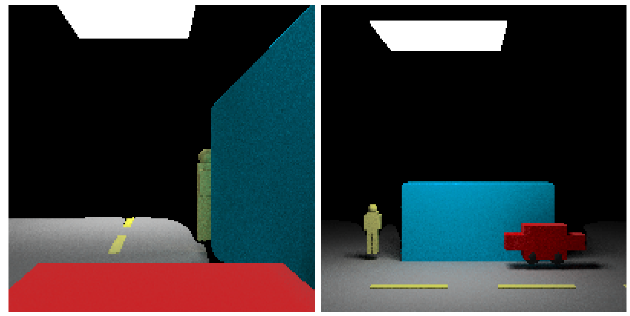

Figure 1: A speeding driver (Left: driver's view) crashes into a pedestrian (yellow) emerging from behind an obstruction (blue). Given a single frame of camera footage (Right), OMEGA$_{\mathrm{C}}$ infers whether driving below the speed limit would have prevented the crash.

1. Introduction

In this paper we introduce OMEGA$_{\mathrm{C}}$: a Turing-universal programming language for causal reasoning. OMEGA$_{\mathrm{C}}$ allows users to automatically derive causal inferences about phenomena modelled through simulation. This contribution focuses on using OMEGA$_{\mathrm{C}}$ to compute counterfactuals – *what-if* causal inferences about the way the world could have been, had things been different.

OMEGA$_{\mathrm{C}}$ programs are simulation models augmented with probability distributions to represent uncertainty. In a similar vein to other probabilistic languages, OMEGA$_{\mathrm{C}}$ provides primitive operators for conditioning, which revises the model to be consistent with observed evidence. Counterfactuals, however, cannot be expressed through probabilistic conditioning alone. They have the form: "Given that some evidence E is true, what would Y have been had X been different?" For example, given that a drug treatment was not effective on a patient, would it have been effective at a stronger dosage? Although one can condition on E being true, attempting to condition on X being different to the value it actually took is contradictory.

In order to express these hypothetical scenarios, OMEGA$_{\mathrm{C}}$ introduces a **do** operator, which constructs *interventions*:

$$Y \mid \mathbf{do}(X \to x) \tag{1}$$

This evaluates to what Y would have been had X been bound to x when Y was defined. Here, X and Y are program variables, typically bound to random variables.

A counterfactual in OMEGA$_{\mathrm{C}}$ is then simply an expression of the form $Y_x \mid E$ where $Y_x = Y \mid \mathbf{do}(X \to x)$, i.e., one that contains both a condition and an intervention, in a particular pattern. The salient feature of this counterfactual pattern is that conditioning on E revises the distribution over Y (because E is defined in terms of Y, not Y_X), and it is to this revised distribution that a causal intervention is performed. The relative nesting of the condition and intervention reflects the fact that we want to intervene Y but not on the evidence E.

To illustrate the potential of counterfactual reasoning within a universal programming language, consider the scenario of an expert witness called to determine, from only a frame of recorded video (Fig. 1), whether a driver was to blame for them crashing into a pedestrian. Using OMEGA$_{\mathrm{C}}$, the expert could first construct a probabilistic model that includes the car dynamics, the driver and pedestrian's behaviour, and a rendering function that produces two dimensional images

[1]CSAIL, MIT, USA [2]Key Lab of High Confidence Software Technologies, Ministry of Education, Department of Computer Science and Technology, Peking University, China. Correspondence to: Zenna Tavares <zenna@csail.mit.edu>, Xin Zhang <xin@pku.edu.cn>.

Proceedings of the 38^{th} International Conference on Machine Learning, PMLR 139, 2021.

from the three dimensional scene. She could then condition the model on the captured images to infer the conditional distribution over the driver's velocity, determining the probability that the driver had been speeding. Next, she could then pose a counterfactual in OMEGA$_\mathrm{C}$, querying whether the crash would have still occurred even if the driver had obeyed the speed limit. If she later wanted to investigate the culpability of another candidate cause, such as the presence of an obstacle occluding the driver's view, she could do so by adding a single-line, and without modifying her underlying models at all.

Causal reasoning is currently done predominantly using *causal graphical models* (21): graphs whose vertices are variables, and whose directed edges represent causal dependencies. Despite widespread use, causal graphs cannot easily express many real-world phenomena. One reason for this is that causal graphs are equivalent to *straight-line programs*: programs without conditional branching or loops – just finite sequences of primitive operations. Straight-line languages are not Turing-complete; they cannot express unbounded models with an unknown number of variables. In practice, they lack many of the features (composite functions, data types, polymorphism, etc.) necessary to express the kinds of simulation models we would like to perform causal inference in.

OMEGA$_\mathrm{C}$, in contrast, can express complex simulation models, but the design of a generic **do** operator presents several challenges. In particular, to construct Y_X, we must be able to copy Y in such a way that the code that defines it is retroactively modified. This goes beyond the capabilities of existing programming languages, probabilistic or otherwise, and hence OMEGA$_\mathrm{C}$ requires a non-standard semantics and implementation.

In summary, we (i) present the syntax and semantics of a universal probabilistic language for counterfactual generative models (Section 3); (ii) provide a complete implementation of OMEGA$_\mathrm{C}$, and (iii) demonstrate counterfactual generative modelling through a number of examples (Section 5). Regarding scope, causal inference includes problems of both (i) inferring a causal model from data, and (ii) given a causal model, predicting the result of interventions and counterfactuals on that model. We focus here on the latter.

2. Overview of Counterfactuals

Counterfactual claims assume some structure is invariant between the original *factual world* and intervened *hypothetical world*. For instance, the counterfactual "If I had trained more, I would have won the match" is predicated on the invariance of the opponent's skill, the existence of the game, laws of physics, etc. Any system for counterfactual reasoning must provide mechanisms to construct hypothetical worlds that maintain invariances (and hence share information) with the factual world, so that for instance the fact that I actually lost the match helps predict whether I would have won the match had I trained harder.

These requirements have been resolved in the context of causal graphical models. Causal interventions are "surgical procedures" which modify single nodes but leave functional dependencies intact. Pearl's *twin-network construction* (21) of counterfactuals duplicates the model into one twice the size. One half is the original model. The other half is a duplicate, modified to express the counterfactual interventions. These halves are joined via a shared dependence on the background facts. Hence, conditioning a variable in the factual world influences the counterfactual world.

To generalize the twin-network construction to arbitrary programs, OMEGA$_\mathrm{C}$ runs two copies of a program, one factual execution, and one counterfactual execution which shares some variables, but where others have been given alternate definitions. It is folklore that programs doing this can be built by hand, but, as in the twin-network construction, each intervention requires writing a separate model, and each counterfactual included doubles the size of the program. The solution in OMEGA$_\mathrm{C}$ is to provide a new **do** operator which removes the need to modify an existing program to add a counterfactual execution. Instead, $t_1 \mid \mathbf{do}(x \to t_2)$ is defined to be the value that a term t_1 would take if x had been set to t_2. This works even if any dependencies of t_1 on x are indirect. For instance, if $y = 2x$, then $y^2 \mid \mathbf{do}(x \to f)$ is equivalent to $(2f)^2$. And note that the variable x can be any variable, even one that is bound to a function, meaning users can compactly define interventions which are substantial modifications. Finally, combining the operator with conditioning automatically gives counterfactual inference.

Our examples show that OMEGA$_\mathrm{C}$ enables compact definition of many counterfactual inference problems. Indeed, we prove that the **do** operator is not expressible as syntactic sugar (as defined by programming language theory).

3. A Calculus for Counterfactuals

Our language OMEGA$_\mathrm{C}$ is a simple functional probabilistic language augmented to support counterfactuals. To achieve this: (1) the syntax includes a **do** operator, and (2) the language evaluation is lazy rather than eager, which is key to handling interventions. In this section, we introduce λ_C, a core calculus of OMEGA$_\mathrm{C}$. After some preliminaries, we show the deterministic semantics of the language, followed by its probabilistic features. Together, intervention and conditioning give the language the ability to do counterfactual inference. Appendix A gives a more formal definition of the entire λ_C language. A Julia implementation of OMEGA$_\mathrm{C}$ can be found at `https://github.com/zenna/Omega.jl`, and a Haskell implementation of λ_C can be found at https://github.com/jkoppel/omega-calculus.

$$\begin{array}{rl} \text{Variables} & x, y, z \in \text{Var} \\ \text{Type } \tau ::= & \text{Int} \mid \text{Bool} \mid \text{Real} \mid \tau_1 \to \tau_2 \\ \text{Term } t ::= & n \mid b \mid r \mid t_1 \oplus t_2 \mid x \mid \textbf{let } x = t_1 \textbf{ in } t_2 \mid \\ & \lambda x : \tau . t \mid t_1(t_2) \mid \textbf{if } t_1 \textbf{ then } t_2 \textbf{ else } t_3 \end{array}$$

Figure 2: Abstract Syntax for λ_C, deterministic fragment

Preliminaries Here, we introduce the notation to describe the semantics of a simple deterministic programming language; Fig. 2 gives the syntax. We use the formalism of operational semantics (24) to describe how one expression reduces to another. Appendix A provides an operational semantics for OMEGA$_C$. Here, we describe these reductions through examples. The execution of an expression is defined both in terms of the expression as well as the current program state. In λ_C, this program state is an environment Γ: a mapping from variables to values.

λ_C has integer numbers (denoted n), Booleans $\{\text{True}, \text{False}\}$ (denoted b), and real numbers (r). $\oplus$ represents a mathematical binary operator such as $+$, $*$, etc. **let** $x = t_1$ **in** t_2 binds variable x to expression t_1 when evaluating t_2. Lambda expressions create functions: $\lambda x.2 * x$ defines a mapping $x \mapsto 2x$.

Next, we show the semantics of operators and **let**. The notation $\left\{\begin{smallmatrix}\Gamma \\ e\end{smallmatrix}\right\}$ denotes a pair of an environment Γ and an expression e, and $\left\{\begin{smallmatrix}\Gamma_1 \\ e_1\end{smallmatrix}\right\} \to \left\{\begin{smallmatrix}\Gamma_2 \\ e_2\end{smallmatrix}\right\}$ denotes that e_1 with environment Γ_1 steps to e_2 with environment Γ_2. For example, in the expression **let** $x = 3$ **in** x, x is first bound to 3, creating a new environment. Finally, x is evaluated by looking up its value in the environment.

$$\left\{\begin{matrix}\Gamma : \emptyset \\ \textbf{let } x = 3 \textbf{ in } x\end{matrix}\right\} \to \left\{\begin{matrix}\Gamma : x \mapsto 3 \\ x\end{matrix}\right\} \to \left\{\begin{matrix}\Gamma : x \mapsto 3 \\ 3\end{matrix}\right\}$$

Function applications are done by substitution, as in other variants of the lambda calculus:

$$\left\{\begin{matrix}\Gamma : \emptyset \\ (\lambda x.(x + x)(2)\end{matrix}\right\} \to \left\{\begin{matrix}\Gamma : \emptyset \\ 2 + 2\end{matrix}\right\} \to \left\{\begin{matrix}\Gamma : \emptyset \\ 4\end{matrix}\right\}$$

The above semantics is *eager*: **let** $x = t_1$ **in** t_2 first evaluates t_1 and then binds the result to x, creating a new environment in which to then evaluate t_2. We next show how this is problematic for counterfactuals. and how we address it using lazy semantics.

Deterministic OMEGA$_C$ OMEGA$_C$ adds a new term: the **do** expression (Fig. 3). $t_1 \mid \textbf{do}(x \to t_2)$ evaluates t_1 to the value that it would have evaluated to, had x been defined as t_2 at its point of definition. Here, x can be any variable that is in scope, bound locally or globally, and t can be any term denoting a value. One idea is to define **do** similarly to **let**: $t_1 \mid \textbf{do}(x \to t_2)$ would rebind x to t_2 when evaluating t_1. However, this does account for transitive dependencies. For example, **let** $x = 0$ **in let** $y = x$ **in** $(y \mid \textbf{do}(x \to 1))$ should evaluate to 1, but by the time we evaluate the **do**, y has already been bound to 0 so that rebinding x does nothing. To overcome this, we redefine **let** to use *lazy evaluation*.

In lazy evaluation, instead of storing the value of a variable in the environment, the execution stores its defining expression as well as the environment when the variable is defined. So, while environments for eager evaluation store mappings $x \mapsto v$ from variable x to value v, in lazy evaluation, the environments store mappings $x \mapsto (\Gamma, e)$, which map each variable x to a *closure* containing both its defining expression e and the environment Γ in which it was defined. A variable, such as x, is evaluated by evaluating its definition under the environment where it is defined.

We can now define **do**: $y \mid \textbf{do}(x \to -1)$ evaluates y under a new environment which is created by recursively mapping all bindings for x in the current environment to -1. This includes both the binding of x at the top level and the bindings in an environment that is used in any closure. The following example demonstrates this process:

$$\begin{aligned}
&\left\{\begin{matrix}\Gamma : \emptyset \\ \textbf{let } x = 0 \textbf{ in let } y = x + 1 \textbf{ in } y + (y \mid \textbf{do}(x \to -1))\end{matrix}\right\} \\
\xrightarrow{1} &\left\{\begin{matrix}\Gamma : x \mapsto (\emptyset, 0) \\ \textbf{let } y = x + 1 \textbf{ in } y + (y \mid \textbf{do}(x \to -1))\end{matrix}\right\} \\
\xrightarrow{2} &\left\{\begin{matrix}\Gamma : x \mapsto (\emptyset, 0), y \mapsto (x \mapsto (\emptyset, 0), x + 1) \\ y + (y \mid \textbf{do}(x \to -1))\end{matrix}\right\} \\
\xrightarrow{3} &\left\{\begin{matrix}\Gamma : x \mapsto (\emptyset, 0), y \mapsto (x \mapsto (\emptyset, 0), x + 1) \\ \left\{\begin{smallmatrix}\Gamma : x \mapsto (\emptyset, 0) \\ x + 1\end{smallmatrix}\right\} + (y \mid \textbf{do}(x \to -1))\end{matrix}\right\} \\
\xrightarrow{4} &\left\{\begin{matrix}\Gamma : x \mapsto (\emptyset, 0), y \mapsto (x \mapsto (\emptyset, 0), x + 1) \\ 1 + (y \mid \textbf{do}(x \to -1))\end{matrix}\right\} \\
\xrightarrow{5} &\left\{\begin{matrix}\Gamma : x \mapsto (\emptyset, 0), y \mapsto (x \mapsto (\emptyset, 0), x + 1) \\ 1 + \left\{\begin{smallmatrix}\Gamma : x \mapsto (\emptyset, 0), y \mapsto (x \mapsto (\emptyset, 0), x + 1) \\ (y \mid \textbf{do}(x \to -1))\end{smallmatrix}\right\}\end{matrix}\right\} \\
\xrightarrow{6} &\left\{\begin{matrix}\Gamma : x \mapsto (\emptyset, 0), y \mapsto (x \mapsto (\emptyset, 0), x + 1) \\ 1 + \left\{\begin{smallmatrix}\Gamma : x \mapsto (\emptyset, -1), y \mapsto (x \mapsto (\emptyset, -1), x + 1) \\ y\end{smallmatrix}\right\}\end{matrix}\right\} \\
\xrightarrow{7} &\left\{\begin{matrix}\Gamma : x \mapsto (\emptyset, 0), y \mapsto (x \mapsto (\emptyset, 0), x + 1) \\ 1 + \left\{\begin{smallmatrix}\Gamma : x \mapsto (\emptyset, -1) \\ x + 1\end{smallmatrix}\right\}\end{matrix}\right\} \\
\xrightarrow{8} &\left\{\begin{matrix}\Gamma : x \mapsto (\emptyset, 0), y \mapsto (x \mapsto (\emptyset, 0), x + 1) \\ 1 + 0\end{matrix}\right\} \\
\xrightarrow{9} &\left\{\begin{matrix}\Gamma : x \mapsto (\emptyset, 0), y \mapsto (x \mapsto (\emptyset, 0), x + 1) \\ 1\end{matrix}\right\}
\end{aligned}$$

Term $t ::= \cdots \mid t_1 \mathbin{|} \mathbf{do}(x \to t_2)$

Figure 3: Abstract Syntax for λ_C, causal fragment

The program is evaluated under an empty environment. (1) Evaluating the outermost **let** binds x to a closure $(\emptyset, 0)$ (consisting of the initial environment and x's definition). (2) y is bound to a closure, containing the environment from step (1) and y's definition. The left operand of the addition is then evaluated, by first (3) looking up its closure in the environment, and then (4) evaluating its definition under the corresponding environment in the closure. To evaluate the **do** in the right operand, (5) the current environment is copied, and then (6) modified to rebind all definitions of x to -1. The right operand of the addition is a do expression of y, which the execution tries to evaluate under the current environment, by (7) looking up the closure of y in this "intervened" environment, and then (8) evaluating it. (9) The final result of the program is then 1.

To implement **do** we introduce a procedure which we call retroactive-updating. Informally, this creates a new environment that rebinds all occurrences of the intervened variable within a closure to its intervened value. This is formally specified with respect to the operational semantics in the supplementary material.

Type $\tau ::= \cdots \mid \Omega$ Term $t ::= \cdots \mid \perp \mid t_1 \mathbin{|} t_2 \mid \mathbf{rand}(t)$

Figure 4: Abstract Syntax for λ_C, probabilistic fragment

Probabilistic OMEGA$_\text{C}$ In probability theory, a random variable is a function from a sample space Ω to some domain τ. λ_C defines random variables similarly: as functions of type $\Omega \to \tau$. This separates the source of randomness of a program from its main body, which allows for a clean definition of counterfactuals.

Fig. 4 shows the abstract syntax of the probabilistic fragment. It introduces a new type Ω, representing the sample space. Ω is left unspecified, save that it may be sampled from uniformly. In most applications, Ω will be a hypercube, with one dimension for each independent sample. To access the values of each dimension of this hypercube, one of the $\oplus$ operators must be the indexing operator [], so that $\omega[i]$ evaluates to the ith component of ω.

Random variables are normal functions. If $\Omega = [0, 1]$, and $a < b$ are integer constants, then $R = \lambda\omega : \Omega.\omega * (b - a) + a$ is a random variable uniformly distributed in $[a, b]$. The **rand** operator then samples from a random variable: **rand** R returns a random value drawn uniformly from $[a, b]$. Note that unlike in other probabilistic languages, we separate the construction of random variables from their sampling. Consequently, **rand** does not occur in the definition of a random variable itself.

To support conditioning, we use $\perp$ to denote the undefined value. Any expression (excluding **rand**) that depends on a $\perp$ value will result in another $\perp$ value. A program execution is invalid if it evaluates to $\perp$.

Conditioning can now be defined as syntactic sugar: $t \mathbin{|} E$ is defined as $\lambda\omega.\mathbf{if}\ E(\omega)\ \mathbf{then}\ t(\omega)\ \mathbf{else}\ \perp$. In words: if the evidence E is false in scenario ω, then $t \mathbin{|} E$ is undefined in that scenario.

One can imagine the execution of a λ_C program as a rejection sampling process: we ignore all samples from **rand** that would make the program evaluate to $\perp$. In the implementation, we use a much more efficient inference algorithm (31).

For example, let $\Omega = \{1, 2, \ldots, 10\}$, and consider the program **rand** $\lambda\omega.\omega * 2 \mathbin{|} \lambda\omega.\omega < 4$. If $\omega \geq 4$, then evaluating the random variable results in $\perp$. The **rand** operator hence runs the variable with ω drawn uniformly from $\{1, 2, 3\}$, resulting in 2, 4, or 6, each with $\frac{1}{3}$ probability.

Counterfactuals A counterfactual is a random variable of the form $(t_1 \mathbin{|} \mathbf{do}(x \to t_2)) \mid E$. Consider the following program depicting a game where a player chooses a number c, and then a number ω is drawn randomly from a sample space $\Omega = \{0, 1, \ldots, 6\}$. He wins iff c is within 1 of ω. The query asks: given that the player chose 1 and did not win, what would have happened, had the player chosen 4?

```
let c = 1 in
let x = λω . if (ω-c)*(ω-c)<= 1
             then 1 else -1 in
let cfx = (x | do(c → 4)) | λω. x(ω) == -1)
  in rand(cfx)
```

As before, the **rand** expression is evaluated in the context $\Gamma_1 = \{c \mapsto (\emptyset, 1), x \mapsto (c \mapsto \ldots, \lambda\omega.\mathbf{if}\ \ldots)\}$. Its argument, a conditioning term, desugars to $\lambda\omega'.\mathbf{if}\ x(\omega') == -1\ \mathbf{then}\ (x \mathbin{|} \mathbf{do}(c \to 4))(\omega')\ \mathbf{else}\ \perp$. This random variable evaluates to $\perp$ for $\omega' \in \{0, 1, 2\}$, so the program is evaluated with ω' drawn uniformly from $\{3, 4, 5, 6\}$. The **do** expression $x \mathbin{|} \mathbf{do}(c \to 4)$ is reduced to evaluating x in the context $\Gamma_2 = \{c = \ldots, x = (c \mapsto (\emptyset, 4), \lambda\omega.\mathbf{if}\ \ldots)\}$. This is then applied to ω', and the overall computation hence evaluates to 1 with probability $\frac{3}{4}$ and -1 with probability $\frac{1}{4}$.

Syntactic Sugar OMEGA$_\text{C}$ introduces some syntactic conveniences on top of λ_C. Random variables are functions but it is convenient to treat them as if they were the values in their domains. To support this, OMEGA$_\text{C}$ interprets the application of a function to one or more random variables *pointwise* – if both X and Y are random variables, then $X+Y$ is also a random variable defined as $\lambda\omega.X(\omega)+Y(\omega)$.

Similarly, if x is a constant, then $X = x$ is $\lambda\omega.X(\omega) = x$. In addition, OMEGA$_C$ represents distribution families as functions from parameters to random variables. For instance, **bern** $= \lambda p.\lambda\omega.\omega[1] < p$ represents the Bernoulli family by mapping a parameter $p \in [0, 1]$ to a random variable that is true with probability p. Finally, since λ_C is purely functional, if $X =$ **bern**(0.5) and $Y =$ **bern**(0.5), then X and Y are not only i.i.d. but the very same random variable, which is not often what we want. OMEGA$_C$ defines the syntax $\sim X$, so that in **let** $X \sim$ **bern**$(0.5), Y \sim$ **bern**(0.5), X and Y are independent.

3.1. Other Composite Queries

Conditioning and intervening can be composed arbitrarily. This allows us to express a variety of causal queries.

To demonstrate, we adapt an example from (21), whereby (i) with probability p, a court orders rifleman A and B to shoot a prisoner, (ii) A's calmness C ranges uniformly from 1 (cool) to 0 (nervous), (iii) if C falls below a threshold q (and hence with probability q) A nervously fires regardless of the order, and (iv) the prisoner dies ($D = 1$) if either shoots. In OMEGA$_C$:

```
let p = 0.7, q = 0.3,
  E = ~ bern(p),        -- Execution order
  C = ~ unif(0, 1),     -- Calmness
  N = C < q,             -- Nerves
  A = E or N,           -- A shoots
  B = E,                -- B shoots on order
  D = A or B in         -- Prisoner Dies
```

As we have seen, counterfactuals condition the real world and consider the implications in a hypothetical world, e.g.:

```
-- Given D, would D be true had A not fired?
(D | do(A → 0)) | D
```

Non-atomic Interventions *Atomic* interventions, which replace a random variable with a constant, often do not reflect the kinds of interventions that have, or even could have, taken place in the real-world. Various non-atomic interventions are easily expressed in OMEGA$_C$:

Conditional interventions (8) replace a variable with a deterministic function of other observable variables:

```
-- if A's nerves had spread to B, would D occur?
D | do(B → C < q)
```

A *mechanism change* (32) alters the functional dependencies between variables.

```
-- Would D occur if it took both shots to kill him?
(D | do(D → A and B)) | D
```

Parametric interventions (9) alter, but do not break, causal dependencies. They are expressible by intervening a variable to be a function of its non-intervened self.

```
-- If A were more calm, would D have occurred?
D | do(C → C * 1.2)
```

Partial compliance (20) is where an intervention fails to have any effect with some probability:

```
-- Would D have occurred had we attempted (and failed
-- with probability s) to prevent A shooting?
D | do(A → if ~ bern(s) then 0 else A)
```

"Fat-hand" interventions (9) inadvertently (and probabilistically) affect some variables other than the intended ones:

```
-- Would D be dead if we stopped A from firing and
-- (with probability r) also prevented B, too?
D | do(A → 0, B → if ~ bern(r)
                    then 0 else B)
```

4. Why do is not Syntactic Sugar

In his influential thesis work, Felleisen (10) addressed the question of when a language construct is mere "syntactic sugar," vs. when it increases a language's power. In this, he provided the notions of *expressibility* and *macro-expressibility*. A language construct F is expressible in terms of the rest of the language if the minimal subprograms containing F can be rewritten to not use F while preserving program semantics. Macro-expressibility further stipulates that these rewrites must be local.

With these, he also provided an ingeniously simple proof technique: a construct is not macro-expressible if there are two expressions which are indistiguishable without the language construct (i.e.: they run the same when embedded into any larger program), but distinguishable with it.

In the following theorem, we prove that we cannot implement the **do** operator as a syntactic sugar (i.e., macro) in the original OMEGA language.

From our literature search, this is also the first time any variant of dynamic scope has been proven not macro-expressible in a language without dynamic scope.

Theorem 1. The **do** operator is not macro-expressible in λ_C without **do**.

Proof. According to the proof technique of Felleisen (10), to show **do** is not macro-expressible in λ_C without **do**, it suffices to find two expressions P and P' such that, for any evaluation context C in λ_C without **do**, $C[P] = C[P']$, but such that there is an evaluation context C in λ_C with **do** such that $C[P] \neq C[P']$.

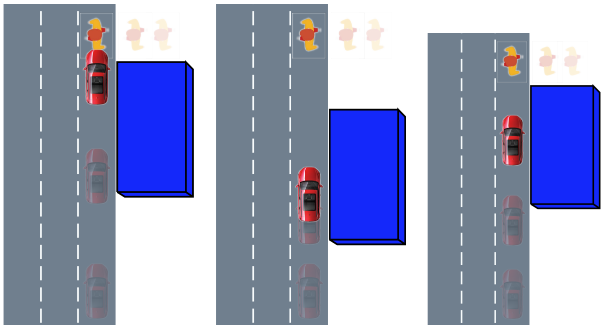

Figure 5: Traces of counterfactual scenarios through time. Each figure is a single sample from (Left) the posterior – the car crashes into the pedestrian, (Middle) the counterfactual on intervening the obstacle position, and (Right) intervening the driver speed. Each image shows the driver and car at (in decreasing transparency) at times 1, 9, and 19.

Let $P = \lambda f.\lambda x.(f\ 0)$, and $P' = \lambda f.(\lambda a.\lambda x.a)(f\ 0)$.

Note that all constructs of λ_C except **do** and **rand** are macro-expressible in terms of the pure lambda calculus. After fixing a random seed, **rand** is also deterministic. Hence, with a fixed seed, λ_C without **do** respects beta equivalence. Hence, since $P \equiv_\beta P'$, for any context C which does not contain **do**, $C[P] = C[P']$.

Now pick:

$$C[e] = ((\lambda g.g\ 0 \mid \mathbf{do}(p \to 1))(e(\lambda x.p))) \mid \mathbf{do}(p \to 0)$$

Then $C[P] \Downarrow 1$, but $C[P'] \Downarrow 0$, where $\Downarrow$ is the reduction relation between terms.

□

5. Experiments

Here we demonstrate counterfactual reasoning in OMEGA$_\text{C}$ through three case studies. All experiments were performed using predicate exchange (31).

Car-Crash Model Continuing from the introduction, this example asks whether a crash would have occurred had a car driven more slowly, given observed camera footage. Let S be the space of scenes, where each scene $s \in S$ consists of the position, velocity, and acceleration of the car, pedestrian and an obstacle. A ray-marching based (1) rendering function $r : S \to I$ maps a scene to an image. The driver acts according to a driver model – a function mapping $s \in S$ to a target acceleration:

```
let
  drivermodel = λ car, ped, obs .
    if cansee(car, ped, obs) -- if ped is visible
    then -9                  -- decelerate
    else 0,                  -- else maintain
```

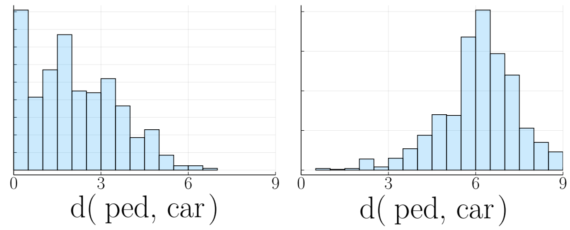

Figure 6: Histograms of causal effect of interventions. How close would the car have come to the pedestrian had (Left) the velocity been reduced to the speed limit (CarV $\to$ 14), or (Right) the obstacle been moved. Even at the speed limit, the driver still would have crashed with high probability.

The expert witness maintains random variables over the car's acceleration, velocity, and position at $t = 0$. The function `simulate` returns state space trajectories of the form $(s_t, s_{t+1}, \ldots, s_n)$. Since the initial scene is a random variable, `Traj` is a random variable over trajectories. Applying `render` to each scene in `Traj` yields a random variable over image trajectories.

```
CarV = ~ normal(12, 4),
CarP = ~ normal(30, 5),
PedV = ~ normal(3, 1),
PedP = ~ normal(1, 2),
InitScene = (CarV, CarP, PedV, PedP, obs),
Traj = simulate(InitScene, drivermodel),
Images = map(render, Traj),
```

We then ask the counterfactual, conditioning the t_{obs}th image on observed data (Figure 1 right) and intervening CarV $\to$ 14.

```
E = (Images[t] == data) and crashed(Traj)
  in (Traj | do(CarV → 14)) | E
```

We can also ask: would the crash have occurred had the obstacle been displaced?

```
in (Traj | do(obs → obs - 3)) | E
```

Figures 5 and 6 visualize the posterior distributions over d(pred, car), the (smallest) distance between the car and the pedestrian.

Glucose Modelling This example queries whether a hypoglycemic episode could have been avoided in a diabetic patient. We first construct an ODE over variables captured in the Ohio Glucose dataset (17): (1) CGM: continuously monitored glucose measurements, (2) Steps: steps walked by patient, (3) Bolus: insulin injection events, and (4) Meals: calorie intake. The recursive function `euler` implements Euler's method to solve the ODE, taking as input an initial state `u` and derivative function `f'`, and producing a time-series $(u_t, u_{t+\Delta t}, u_{t+2\Delta t}, \ldots, u_{tmax})$.

```
let t0 = 0, Δt = 0.1, tmax = 1,
  τ = λ u, t . u,            -- to intervene u
  euler = λ f', u, t .
    let u = τ(u, t), tnext = t + Δt in
      if t < tmax
      then let unext = u + f'(tnext, u) * Δt
        in cons(u, euler(f', unext, tnext))
      else u,
```

We pre-trained a neural network for the derivative function, and added normally distributed noise to the weights to introduce uncertainty, yielding `F'`, a random variable over functions. Given `F'` as input, `euler` produces a random variable over time-series.

```
Series = euler(F', u, t0),
```

Now we can ask, had we eaten (increased food) at $t = 0.2$, would the hypoglycemic event have occurred? We use the function τ to intervene. It maps `u` at every time `t` to a new value, since `u` is internal to `euler`.

```
τint = λ u, t. if t == 0.5
         then [u[1], u[2], inc(u[3])] else u,
Series = Series | do(τ → τint),
```

As a more exotic example, suppose we are told that someone has intervened, and hypoglycemia was avoided, but we do not know when the intervention occurred. We construct a distribution over the intervention time, then condition the intervened world to find the posterior over times.

```
CGM = first(Series),
-- Hypoglycemia occurs if x is low at any time
Hypo = any(map( λ x . x < thresh, CGM))
-- Prior over time of intervention
T = ~ unif(0, 1)
-- intervention increases food at time T
τint2 = λ ω . λ u, t. if t == T(ω)
      then [u[1], u[2], inc(u[3])] else u
-- Condition on  hypoglycemia not occurring
-- in intervened world
Hypoint = λ ω .
    (Hypo | do(τ → τint2(ω)))(ω)
  in CGM | ¬ Hypoint
```

As shown in Figure 7(c), it is more plausible that the intervention occurred early in the day.

Counterfactual Planning Consider a dispute between three hypothetical islands (Figure 10): S (South), E (East) and N (North). The people of S consider a barrier between S and N, asking the counterfactual: given observed migration patterns, how would they differ had a border existed.

We model this as a population of agents each acting according in accordance to a Markov Decision Process (25) (MDP) model. Each grid cell is a state in a state space $\mathcal{S} = \{(i, j) \mid i = 1 \ldots 7, j = 1 \ldots 6\}$. The action space moves an agent a single cell: $\mathcal{A} = \{\text{up}, \text{down}, \text{left}, \text{right}\}$. Each agent acts according to a reward function that is a function of the state they are in only $R : \mathcal{S} \to \mathbb{R}$. This reward function is normally distributed, conditional on the country the agent originates from. For $t = 100$ timesteps we simulate the migration behavior of each individual using value iteration and count the amount of time spent in each country over the time period. Figure 8 shows population counts according to these dynamics. Figure 9 shows migration in the prior, after conditioning on an observed migration pattern (constructed artificially), and the counterfactual cases (adding the border).

But-for Causality in Occlusion In this experiment, we implement "but-for" causation (13) to determine (i) whether a projectile's launch-angle is the cause of it hitting a ball, and (ii) occlusion, i.e. whether one object is the cause of an inability to see another. An event C is the but-for cause of an event E if had C not occurred, neither would have E (12). But-for judgements cannot be resolved by conditioning on the negation of C, since this fails to differentiate cause from effect. Instead, the modeler must find an alternative world where C does not hold. In OMEGA$_\text{C}$, a value $\omega \in \Omega$ encompasses all the uncertainty, and hence we define but-for causality relative to a concrete value ω.

Definition 1. Let $C_1, \ldots C_n$ be a set of random variables and $c_1, \ldots, c_n$ a set of values. With respect to a world ω, the conjunction $C_1 = c_1 \wedge \cdots \wedge C_n = c_n$ is the but-for cause of a predicate $E : \Omega \to \text{Bool}$ if (i) it is true wrt ω and (ii) there exist $\hat{c}_1, \ldots, \hat{c}_n$ such that:

$$(E \mid \mathbf{do}(C_1 \to \hat{c}_1, \ldots, C_n \to \hat{c}_n))(\omega) = \text{False} \quad (2)$$

$E(\omega) = \text{True}$ is a precondition, the effect must actually have occurred for but-for to be defined.

But-for is defined existentially. To solve it, OMEGA$_\text{C}$ relies on predicate relaxation (31), which underlies inference in OMEGA$_\text{C}$. That is, E is a predicate that in (i) is true iff the projectile hits the ball, and in (ii) is true iff the yellow object is occluded in the scene, computed by tracing rays from the viewpoint and checking for intersections. Predicate relaxation transforms E into soft predicate $\tilde{E}$ which returns a value in $[0, 1]$ denoting how

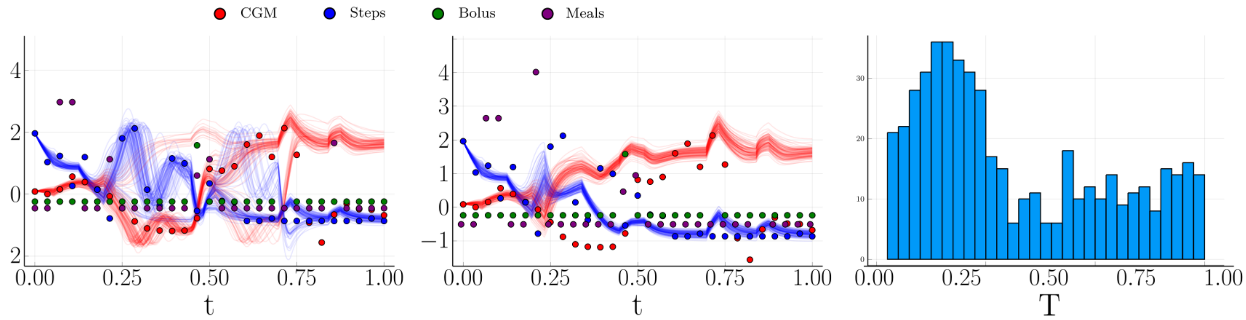

Figure 7: Glucose time series model. Dots are datapoints, trajectories sampled from prior. (Left) Prior samples, (Middle) Samples from interventional distributions under Meal $\to 5$ at $t = 0.20$, (Right) Posterior over time T of intervention given hypoglycemia did not occur after intervention.

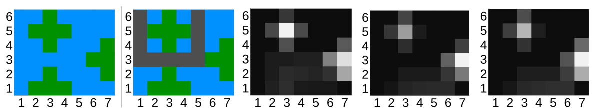

Figure 8: Map (i) without / (ii) with boundary. Sample from population counts after n timesteps of MDP based migration. (iii) unconditional sample, (iv) conditional sample (v) counterfactual sample.

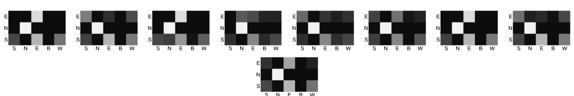

Figure 9: Three samples of migration under three conditions. Each figure shows the migration from islanders born in S, N, or E (y-axis) to S, N, E, W (water) or B (barrier) on the x-axis. We accumulate all states visited in each persons' trajectory. (Plots 1 to 3 from left) Prior samples, (4 to 6) Conditioned on observations, (7 to 9) counterfactual: conditioned and with intervention (border).

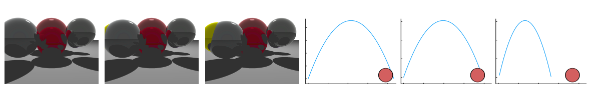

Figure 10: But-for causality. Left to Right: stages of optimization to infer that grey-sphere is cause of inability to see yellow sphere, and launch-angle is cause of projectile colliding with ball.

close E is to being satisfied. Using this, our implementation uses gradient descent over $\hat{c}_1, \ldots, \hat{c}_n$ to minimize $(\tilde{E} \mid \mathbf{do}(C_1 \to \hat{c}_1, \ldots, C_n \to \hat{c}_n))(\omega)$. In (i) $\hat{c}_i$ is the launch-angle and in (ii) $\hat{c}_{x,y,z}$ is the position of the occluder. Finding $\hat{c}_i$ such that soft $E(\hat{c}_i) = 0$ confirms a but-for cause. In Figure 10 we present a visualization of the optimization, which ultimately infers that the angle is the cause of collision and the grey-sphere is the cause of the viewer's inability to see the yellow sphere.

6. Related Work and Discussion

Related work. Operators resembling **do** appear in existing PPLs. Venture (16) has a force expression `[FORCE <expr> <value>]` which modifies the current *trace* object (a mapping from random primitives to values) so that the simulation of `<expr>` takes on the value `<value>`. It is intended as a tool for initialization and debugging. Pyro (5) and Anglican (34) have similar mechanisms. This can and has (18; 23) been used to compute counterfactuals by (i) approximating the posterior with samples, (ii) revising the model with an intervention, and then (ii) simulating the intervened model using the posterior samples instead of priors.

The fundamental distinction is that in OMEGA$_\mathrm{C}$, the operators to condition and intervene both produce new random variables, which can then be further conditioned or intervened to produce *counterfactual variables*, which in turn can be either sampled from or reused in some other process. The Pyro approach, in contrast, computes *counterfactual queries* by performing inference first and then changing the model second. This has several practical consequences. Counterfactual queries in OMEGA$_\mathrm{C}$ tend to be significantly more concise, and require none of the manual hacks. More fundamentally, OMEGA$_\mathrm{C}$ does not embed an inference procedure into the counterfactual model itself, which muddles the distinction between modelling and inference. In this vein, Pyro is similar to Metaverse (23), a recent Python based system, which mirrors Pearl's three steps of abduction, action and prediction, using importance sampling for inference. A downside of this approach is that it is difficult to create the kinds of composite queries we have demonstrated. We explore this in more detail in the Appendix.

RankPL (27) uses ranking functions in place of numerical probability. It advertises support for causal inference, as a user can manually modify a program to change a variable definition. Baral et al. (4) described a recipe to encode counterfactuals in P-log, a probabilistic logic programming language. However, no language construct is provided to automate this process, which they call "intervention". There has also been work in adding causal operators to knowledge-based programming (11), answer-set programming (7), and logic programming (22). There are several libraries for causal inference on causal graphs (6; 28; 33; 3; 2). Whittemore (6) is an embedded Clojure DSL implementing the do-calculus (21). It can estimate the results of interventions, but not counterfactuals, from a dataset.

Ibeling and Icard (14) introduce computable structural equation models (SEMs) to support infinite variable spaces, and prove that an axiomatization of counterfactuals is sound and complete. OMEGA$_C$ similarly supports open-world models, but our approach is constructive rather than axiomatic – we provide primitives to construct and compute counterfactuals. Ness et al. (19) relates SEMs to Markov process models, which are naturally expressible in OMEGA$_C$. They introduce a novel kind of intervention that finds a change to induce a target post-equilibrium value. A version of this is expressible within OMEGA$_C$– first construct a distribution over interventions, then condition that distribution on the target post-equilibrium value occurring.

Alternative approaches. Some languages have inbuilt mechanisms for *reflection* – the ability to introspect and dynamically execute code, Python for instance includes `getsource(foo)` which returns the source code of a function `foo`. By extracting the source code of a model, transforming it, and reexecuting the result with `eval`, a system of interventions could be formulated. This could be a useful way to bring counterfactuals to existing languages such as Python which cannot support lazy evaluation.

While we have presented a minimal language here, OMEGA$_C$ is also implemented in Julia. Since Julia is not lazy, it is less flexible than OMEGA$_C$, suffering some of the limitations of Pyro. We detail this in the Appendix.

Invariants in counterfactuals. An important property of counterfactual inference is that observations in the factual world carry over to the counterfactual world. This property is easy to satisfy in conventional causal graphs as all exogenous and endogenous variables are created and accessed statically. However, this is not true in OMEGA$_C$ as variable creation and access can be dynamic. Concretely, interventions can change the control-flow of a program, which in turn can cause mismatches between variable accesses in the factual world and ones in the counterfactual world. To address this issue, we tie variable identities to program structures. Appendix B discusses this in detail.

Limitations. Procedures such as the PC algorithm (30) handle situations where a causal relationship exists, but nothing is known about the relationship other than that it is an arbitrary function. Like other probabilistic programming languages, OMEGA$_C$ cannot reason about such models.

In some cases the variable we want to intervene is internal to some function and not in scope at the point where we want to construct an intervention. In other cases, the value we want to intervene (e.g. `(x + 2)` in `2*(x + 2)` is not bound to a variable at all. While it is always possible to manually modify the program to expose these inaccessible values, future work is to increase the expressiveness of OMEGA$_C$ to be able to automatically intervene in such cases. Since our formalism relies on variable binding, this would require an entirely different mechanism to what we have presented.

References

[1] Differentiable Path Tracing on the TPU. https://blog.evjang.com/2019/11/jaxpt.html. Accessed: 2021-01-01.

[2] ggdag. https://ggdag.malco.io/. Accessed: 2019-03-08.

[3] pgmpy. http://pgmpy.org/. Accessed: 2019-03-08.

[4] Chitta Baral and Matt Hunsaker. Using the Probabilistic Logic Programming Language P-log for Causal and Counterfactual Reasoning and Non-Naive conditioning. In *IJCAI 2007, Proceedings of the 20th International Joint Conference on Artificial Intelligence, Hyderabad, India, January 6-12, 2007*, pages 243–249, 2007.

[5] Eli Bingham, Jonathan P. Chen, Martin Jankowiak, Fritz Obermeyer, Neeraj Pradhan, Theofanis Karaletsos, Rohit Singh, Paul Szerlip, Paul Horsfall, and Noah D. Goodman. Pyro: Deep Universal Probabilistic Programming. *Journal of Machine Learning Research*, 2018.

[6] Joshua Brulé. Whittemore: An Embedded Domain Specific Language for Causal Programming. *arXiv preprint arXiv:1812.11918*, 2018.

[7] Pedro Cabalar. Causal Logic Programming. In *Correct Reasoning*, pages 102–116. Springer, 2012.

[8] Juan Correa and Elias Bareinboim. A Calculus for Stochastic Interventions: Causal Effect Identification and Surrogate experiments. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 10093–10100, 2020.

[9] Frederick Eberhardt and Richard Scheines. Interventions and Causal Inference. *Philosophy of Science*, 74(5):981–995, 2007.

[10] Matthias Felleisen. On the Expressive Power of Programming Languages. In *ESOP'90, 3rd European Symposium on Programming, Copenhagen, Denmark, May 15-18, 1990, Proceedings*, pages 134–151, 1990.

[11] Joseph Halpern and Yoram Moses. Using Counterfactuals in Knowledge-Based Programming. volume 17, pages 97–110, 07 1998.

[12] Joseph Y Halpern and Christopher Hitchcock. Actual Causation and the Art of Modeling. *arXiv preprint arXiv:1106.2652*, 2011.

[13] Daniel M Hausman, Herbert a Simon, et al. *Causal Asymmetries*. Cambridge University Press, 1998.

[14] Duligur Ibeling and Thomas Icard. On Open-Universe Causal Reasoning. In *Uncertainty in Artificial Intelligence*, pages 1233–1243. PMLR, 2020.

[15] Oleg Kiselyov, Chung-chieh Shan, and Amr Sabry. Delimited Dynamic Binding. In *Proceedings of the 11th ACM SIGPLAN International Conference on Functional Programming, ICFP 2006, Portland, Oregon, USA, September 16-21, 2006*, pages 26–37, 2006.

[16] Vikash Mansinghka, Daniel Selsam, and Yura Perov. Venture: A Higher-Order Probabilistic Programming Platform with Programmable Inference. *arXiv preprint arXiv:1404.0099*, 2014.

[17] Cindy Marling and Razvan C Bunescu. The OhioT1DM Dataset for Blood Glucose Level Prediction. In *KHD@ IJCAI*, 2018.

[18] Robert Ness. Lecture Notes for Causality in Machine Learning, Section 9.6: “Bayesian counterfactual algorithm with SMCs in Pyro”, 2019.

[19] Robert Osazuwa Ness, Kaushal Paneri, and Olga Vitek. Integrating Markov processes with Structural causal Modeling Enables Counterfactual Inference in Complex Systems. *arXiv preprint arXiv:1911.02175*, 2019.

[20] Judea Pearl. From Bayesian networks to Causal Networks. In *Mathematical models for handling partial knowledge in artificial intelligence*, pages 157–182. Springer, 1995.

[21] Judea Pearl. *Causality*. Cambridge University Press, 2009.

[22] Luís Moniz Pereira and Ari Saptawijaya. Agent Morality via Counterfactuals in Logic Programming. In *Proceedings of the Workshop on Bridging the Gap between Human and Automated Reasoning - Is Logic and Automated Reasoning a Foundation for Human Reasoning? co-located with 39th Annual Meeting of the Cognitive Science Society (CogSci 2017), London, UK, July 26, 2017.*, pages 39–53, 2017.

[23] Yura Perov, Logan Graham, Kostis Gourgoulias, Jonathan Richens, Ciaran Lee, Adam Baker, and Saurabh Johri. Multiverse: Causal Reasoning Using Importance Sampling in Probabilistic Programming. In *Symposium on advances in approximate bayesian inference*, pages 1–36. PMLR, 2020.

[24] Gordon D Plotkin. A Structural Approach to Operational Semantics. 1981.

[25] Martin L Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. John Wiley & Sons, 2014.

[26] Jarrett Revels, Valentin Churavy, Tim Besard, Lyndon White, Twan Koolen, Mike J Innes, Nathan Daly, Rogerluo, Robin Deits, Morten Piibeleht, Moritz Schauer, Kristoffer Carlsson, Keno Fischer, and Chris de Graaf. jrevels/cassette.jl: v0.3.3, April 2020.

[27] Tjitze Rienstra. RankPL: A Qualitative Probabilistic Programming Language. In *European Conference on Symbolic and Quantitative Approaches to Reasoning and Uncertainty*, pages 470–479. Springer, 2017.

[28] Amit Sharma and Emre Kiciman. DoWhy: Making Causal Inference Easy. `https://github.com/Microsoft/dowhy`, 2018.

[29] Yehonathan Sharvit. Lazy Sequences are not Compatible with Dynamic Scope. `https://blog.klipse.tech/clojure/2018/12/25/dynamic-scope-clojure.html`, 2018.

[30] Peter Spirtes, Clark N Glymour, Richard Scheines, David Heckerman, Christopher Meek, Gregory Cooper, and Thomas Richardson. *Causation, Prediction, and Search*. MIT press, 2000.

[31] Zenna Tavares, Javier Burroni, Edgar Minaysan, Armando Solar Lezama, and Rajesh Ranganath. Predicate Exchange: Inference with Declarative Knowledge. In *International Conference on Machine Learning*, 2019.

[32] Jin Tian and Judea Pearl. Causal Discovery from Changes. *arXiv preprint arXiv:1301.2312*, 2013.

[33] Santtu Tikka and Juha Karvanen. Identifying Causal Effects with the R Package causaleffect. *Journal of Statistical Software*, 76(1):1–30, 2017.

[34] David Tolpin, Jan-Willem van de Meent, and Frank Wood. Probabilistic Programming in Anglican. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 308–311. Springer, 2015.